Quality Attribute-Guided Evaluation of NoSQL Databases: An Experience Report

John Klein, Software Engineering Institute
Ian Gorton, Software Engineering Institute
Neil Ernst, Software Engineering Institute
Patrick Donohoe, Software Engineering Institute
Kim Pham, Solutions IT
Chrisjan Matser, Codespinner, Inc.

Abstract

NoSQL is a family of database technologies that promises to provide unprecedented levels of performance, scalability and availability by simplifying data models and supporting horizontal scaling and data replication. Each NoSQL product embodies a particular set of consistency, availability and partition-tolerance (CAP) tradeoffs, along with a data model that reduces the conceptual mismatch between data access and data storage models. For software developers, the choice of a NoSQL technology imposes a specific distributed software architecture and data model, making the technology selection one that is difficult to defer in a software project. This means technology selection must be done early, often with limited information about specific application requirements, and the decision must balance speed with precision as the NoSQL solution space is large and evolving rapidly. In this paper we present the method and results of a study we performed to compare the characteristics of 3 NoSQL databases for use in healthcare. We describe the aims of the study, the experimental method, and the outcomes of both quantitative and qualitative comparisons of MongoDB, Cassandra and Riak. We conclude by reflecting on some of the fundamental difficulties of performing detailed technical evaluations of NoSQL databases specifically, and big data systems in general, that have become apparent during our study.

Introduction

The exponential growth of data in the last decade has fueled a new specialization for software technology, namely that of data-intensive, or big data, software systems [Agarwal 2011]. Internet-born organizations such as Google and Amazon are at the cutting edge of this revolution, collecting, managing, storing, and analyzing some of the largest data repositories that have ever been constructed. Their pioneering efforts [DeCandia 2007, Chang 2008], along with those of numerous other big data innovators, have made a variety of open source and commercial data management technologies available for any organization to exploit to construct and operate massively scalable, highly available data repositories.

Data-intensive systems have long been built on SQL database technology, which relies primarily on vertical scaling – faster processors and bigger disks – as workload or storage requirements increase. Inherent vertical scaling limitations of SQL databases [Sadalage 2012] have led to new products that relax many core tenets of relational databases. Strictly-defined normalized data models, strong data consistency guarantees, and the SQL standard have been replaced by schema-less, data models, weak consistency, and proprietary APIs that expose the underlying data management mechanisms to the programmer. These "NoSQL" products [Saladage 2012] are typically designed to scale horizontally across clusters of low cost, moderate performance servers. They achieve high performance, elastic storage capacity, and availability by replicating and partitioning data sets across the cluster. Prominent examples of NoSQL databases include Cassandra, Riak, and MongoDB.

Distributed databases have fundamental quality constraints, defined by Brewer's CAP Theorem [Brewer 2012]. When a network partition occurs ("P"- arbitrary message loss between nodes in the cluster), a system must trade consistency ("C" - all readers see the same data) against availability ("A" - every request receives a success/failure response). A practical interpretation of this theorem is provided by Adabi's PACELC [Abadi 2012] which states that if there is a partition (P), a system must trade availability (A) against consistency (C); else (E) in the usual case of no partition, a system must trade latency (L) against consistency (C).

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
18 OCT 2014	**N/A**	

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Quality Attribute-Guided Evaluation of NoSQL Databases: An Experience Report	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
John Klein Ian Gorton /Neil Ernst, Patrick Donohoe, Kim Pham, Chrisjan Matser	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited.

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**SAR**	**21**	

Each NoSQL product embodies a specific set of quality attribute tradeoffs, especially in terms of their performance, scalability, availability, and consistency. Architects must diligently evaluate candidate database technologies and select databases that can satisfy application requirements. This complex and ever-expanding database landscape creates both opportunities - to utilize new technologies to build highly scalable data repositories at lower costs – and also challenges, as an inappropriate database selection can lead to project overruns and even failure. Various benchmark results [7] - [9] give some aid in database selection. However benchmarks are highly generalized, bearing little or no resemblance to the needs for a database in an organization's specific application.. They also give no insights into a whole range of database features that are crucial to a successful deployment, for example handling failures, security, handling different data types, and administration capabilities.

In this paper, we describe a quantitative and qualitative study we carried out for a healthcare provider wishing to adopt NoSQL technologies. We describe the study context, our experimental approach, and the results of both extensive performance and scalability testing and a detailed feature comparison. We also reflect on our experience, describing some of the fundamental problems of NoSQL evaluations that all organizations must face. The contributions of the paper are as follows:

1. A rigorous experimental method that organizations can follow to evaluate the performance and scalability of NoSQL databases
2. Performance and scalability results that empirically demonstrate significant variability in the capabilities of the databases we tested to support the requirements of our healthcare customer.
3. Practical insights and recommendations that organizations can follow to help streamline a NoSQL database comparison for their own applications.

Overview of NoSQL and Big Data Systems

Big data applications are rapidly becoming pervasive across a wide range of business domains. Two examples are:

1. Modern commercial airliners produce approximately 0.5TB of operational data per flight [Finnegan 2013]. This data can be used to diagnose faults, optimize fuel consumption, and predict maintenance. Airlines must build scalable systems to capture, manage, and analyze this data to improve reliability and reduce costs.
2. Big data analytics could save an estimated $450 billion in the USA [Groves 2013]. Analysis of petabytes of data across patient populations, taken from diverse sources such as insurance payers, public health, and clinical studies, can extract new insights for disease treatment and prevention, and reduce costs by improving patient outcomes and operational efficiencies.

Across these and many other domains, big data systems share common requirements that drive the design of suitable software solutions. These include write-heavy workloads [Argawal 2011], high availability [Armbrust 2010], handling workload bursts, and masking component failures while still satisfying service level agreements [Gorton 2014].

The rise of big data applications has caused significant flux in database technologies. While mature relational database technologies continue to evolve, a spectrum of databases labeled "NoSQL" has emerged in the past decade. The relational model imposes a strict schema, which inhibits data evolution and causes difficulties scaling across clusters. In response, NoSQL databases have adopted simpler data models. Common features include schemaless records, allowing data models to evolve dynamically, and horizontal scaling, by sharding and replicating data collections across large clusters. Fig. 1 illustrates the four most prominent data models, and we summarize their characteristics below. More comprehensive information can be found at http://nosql-database.org/.

- *Document databases* store collections of objects, typically encoded using JSON or XML. Documents have keys, and secondary indexes can be built on non-key fields. Document formats are self-describing, and a collection may include documents with different formats. Leading examples are MongoDB (http://www.mongodb.org/) and CouchDB (http://couchdb.apache.org/).
- *Key-value databases* implement a distributed hash map. Records can only be accessed through key searches, and the value associated with each key is treated as opaque, requiring reader interpretation. This simple model facilitates sharding and replication to create highly scalable and available systems. Examples are Riak (http://riak.basho.com/) and DynamoDB (http://aws.amazon.com/dynamodb/).
- *Column-oriented databases* extend the key-value model by organizing keyed records as a collection of columns, where a column is a key-value pair. The key becomes the column name, and the value can be an arbitrary data type such as a JSON document or a binary image. A collection may contain records that have different numbers of columns. Examples are HBase (http://hadoop.apache.org/) and Cassandra (https://cassandra.apache.org/).
- *Graph databases* organize data in a highly connected structure, typically some form of directed graph. They can provide exceptional performance for problems involving graph traversals and sub-graph matching. As efficient

graph partitioning is an NP-hard problem, these databases tend to be less concerned with horizontal scaling, and commonly offer ACID transactions to provide strong consistency. Examples include Neo4j (http://www.neo4j.org/) and GraphBase (http://graphbase.net).

Document Store

"id": "1" "Name": "John" "Employer": "SEI"
"id": "2" "Name": "Ian" "Employer": "SEI" "Previous": "PNNL"

Key-Value Store

"key": "1" value { "Name": "John" "Employer": "SEI"}
"key": "2" value {"Name": "Ian" "Employer": "SEI" "Previous": "PNNL"}

Column Store

"row": "1" , "Employer" "Name"
"SEI" "John"
"row": "2" "Employer" "Name" "Previous"
"SEI" "Ian" "PNNL"

Graph Store

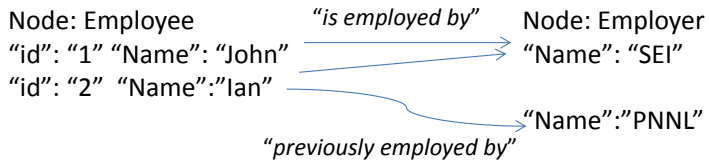

Fig. 1. Examples of Major NoSQL data models

NoSQL technologies have many implications for application design. As there is no equivalent of SQL, each technology supports its own specific query mechanism. These typically make the application programmer responsible for explicitly formulating query executions, rather than relying on query planners that execute queries based on declarative specifications. The ability to combine results from different data collections also becomes the programmer's responsibility. This lack of the ability to perform JOINs forces extensive denormalization of data models so that JOIN-style queries can be efficiently executed by accessing a single data collection. When databases are sharded and replicated, it further becomes the programmer's responsibility to manage consistency when concurrent updates occur, and design applications to tolerate stale data due to latency in update replication. Hence these fundamental differences between NoSQL data models, query languages and distribution architectures make it difficult to migrate both data and applications between databases if a selected NoSQL technology does not meet application requirements. This significantly increases the importance of NoSQL technology selection.

Evaluation Method

Our customer was a large healthcare provider who was developing a new electronic healthcare record (EHR) system to replace their existing system, which used thick client user-facing applications running at sites around the world, all connected to a centralized relational database. They were considering NoSQL technology for two candidate uses: The primary data store for the EHR system, or as a cache local to each site.

We began the engagement with a stakeholder workshop, using a tailored version of the Quality Attribute Workshop Method [Barbacci 2003], to elicit and understand key functional and quality attribute requirements that would guide our technology evaluation. These requirements slotted in to two broad categories, one quantitative, one qualitative, as follows:

1. **Performance/Scalability:** A key requirement was the ability to easily distribute and replicate data across geographically distributed databases, and to achieve high availability and low latencies under load in distributed database deployments. Hence understanding the inherent performance and scalability that is achievable with each candidate NoSQL database was an essential part of the evaluation.

2. **Data Model Mapping:** How would each of the NoSQL data model styles support the health care data model used by our customer? Investigating how the data model could be designed and queried in each different database would provide deep insights into the suitability of the candidate technologies for the customer's applications. This required us to assess the specific features of each database that were appropriate for data modeling and querying, as well as maintaining consistency in a distributed, replicated database.

Many of the key features of NoSQL technologies can be discerned and compared through a detailed analysis of their documentation. However, a thorough evaluation and comparison requires prototyping with each technology to reveal the performance and scalability that each technology is capable of providing [Gorton 2003]. To this end, we developed and performed a systematic experiment that provides a foundation for an "apples to apples" comparison of the 3 technologies we have evaluated. The approach we have taken is as follows:

1. Define driving use cases for the EHR system. The detailed logical data model and collection of read and write queries that will be performed upon the data will follow from the use cases.
2. Define a consistent test environment for evaluating each database, including server platform, test client platform, and network topology.
3. Map the logical data patient records data model to each database's physical data model and load the resulting database with a large collection of synthetic test data.
4. Create a load test client that implements the require use case in terms of the defined mix of read and write operations. This client is capable of simulating many simultaneous requests on the database so that we analyze how each technology responds in terms of performance and scalability as the request load increases.
5. Define and execute test scripts that exert a specified load on the database using the test client.
6. For each defined test case, we have experimented with a number of configurations for each database so we can analyze performance and scalability as each database is sharded and replicated. These deployment scenarios range from a single server for baseline testing up to 9 server instances that shard and replicate data.

Based on this approach, we are able to produce a consistent set of test results from experiments that can be used to assess the likely performance and scalability of each database in an EHR system. The rest of this section describes the setup work we did for steps 1 through 5, and the results of step 6 are presented and discussed in a subsequent section.

Step 1 – Driving Use Cases

We addressed step 1 of our method using the workshop results, to select two use cases to drive our analysis and prototyping. The first use case was retrieving recent medical test results for a particular patient, which is a core EHR function used to populate the user interface whenever a clinician selects a new patient. The second use case was achieving strong consistency for all readers when a new medical test result is written for a patient, because all clinicians using the EHR to make patient care decisions should see the same information about that patient, whether they are at the same site as the patient, or providing telemedicine support from another location.

We chose a subset of the HL7 Fast Healthcare Interoperability Resources (FHIR) data model [HL7 2014] for our analysis and prototyping. The logical data model consisted of FHIR Patient Resources (which include demographic information such as names, addresses, and telephone numbers), and laboratory test results represented as FHIR Observation Resources (which include test type, result quantity, and result units). There was a one-to-many relation from each patient to the associated test results. Although this was a relatively simple model, the internal complexity of the FHIR Patient Resource, with multiple addresses and phone numbers, along with the one-to-many relation from patient to observations, required a number of data modeling design decisions and trade offs in the data mapping performed in Step 3.

Step 2 – Test Environment

All testing was performed using the Amazon EC2 cloud (http://aws.amazon.com/ec2/). Database servers were run on "m1.large" instances, with the database data files and database log files each stored on separate EBS volumes attached to each server instance. The EBS volumes were "standard", not provisioned with the EC2 IOPS feature, to minimize the tuning parameters used in each test configuration. Server instances ran the CentOS operating system (http://www.centos.org).

The test client was also run on an "m1.large" instance, and also used the CentOS operating system.

All instances were in the same EC2 availability zone (i.e. the same cloud data center). Wide-area Network (WAN) topologies were simulated for our partition tolerance testing using the standard Linux `tc` and `iptables` tools.

Step 3 – Mapping the data model and load data set

A synthetic data set was used for testing. This data set contained one million patient records, and 10 million lab result records. The number of lab results for a patient ranged from zero to 20, with an average of seven. As noted above, patient records were represented using the Patient Resource in the FHIR model, and lab result records were represented using the Observation Resource in the FHIR model. Both Patient and Observation Resources were mapped into the data model for each of the databases that were tested. The results of this mapping were of primary interest to our customer, and are presented in the subsequent section below.

Step 4 – Create load test client

Our test client was based on the YCSB framework [Cooper 2010], which provides capabilities for test execution and test measurement. For test execution, YCSB has default data models, data sets, and workloads. We replaced these with implementations specific to our prototype design, however we did use the YCSB's built-in parameters to specify the total number of operations to be performed and the percentage of read operations and write operations in the workload. The test execution capabilities allow the use of multiple execution threads to create multiple concurrent client sessions.

In the measurement framework, for each operation performed, YCSB measures the *operation latency*, which is the time from when the request is sent to the database until the response is received back from the database. The YCSB reporting framework records the minimum, maximum, and average operation latency separately for read and write operations.

YCSB also aggregates the latency measurements into histogram buckets, with separate histograms for read and write operations. There are 1001 buckets: the first bucket counts operations with latency from 0-999 microseconds, the second bucket counts operations with latency from 1000-1999 microseconds, up to bucket 999 which counts operations with latency from 999,000-999,999 microseconds. The final "overflow" bucket counts operations with latency greater than 1,000,000 microseconds (1 second). At the completion of the workload execution, YCSB calculates an approximation to the 95th percentile and 99th percentile operation latency by scanning the histogram until 95% and 99% of the total operation count is reached, and then reporting the latency corresponding to that bucket where the threshold is crossed. There is no interpolation. If the overflow bucket contains more than 5% of the measurements, then the 95th percentile is reported as "0", and if the overflow bucket contains more than 1% of the measurements, then the 99th percentile is reported as "0".

We extended the YCSB reporting framework to report *Overall Throughput*, in operations per second. This measurement was calculated by dividing the total number of operations performed (read plus write) by the workload execution time. The execution time was measured from the start of the first operation to the completion of the last operation in the workload execution, and did not include initial setup and final cleanup times. This execution time was also reported separately as *Overall Run Time*.

Each workload execution produced a separate output file containing measurements and metadata. Metadata was produced by customizing YCSB to record:

- The number of client threads
- The workload identifier
- The database cluster configuration (number of server nodes and IP addresses)
- Workload configuration parameters used for this execution
- The command line parameters used to invoke this execution
- Basic quality indicators (number of operations performed and number of operations completed successfully).
- The time of day of the start and end of this execution.

The measurements were written by the standard YCSB reporting framework in JSON format.

Step 5 – Define and execute test scripts

Our test scripts were split into two sections: transactional workload and partition tolerance.

Our stakeholder workshop identified that the typical transactional workload for the EHR system was 80% read operations, and 20% write operations. For our driving use cases, we defined a read/write workload that comprised a mix of 80% read operations, each retrieving the five most recent observations for a single patient, and 20% write operations, each inserting a single observation record for a single patient.

Our stakeholders were also interested in using the NoSQL technology as a local cache, so we defined a write-only workload that was representative of a daily preload of a local cache from a centralized primary data store with records for

patients with scheduled treatment appointments for that day. Finally, we defined a read-only workload that was representative of flushing the cache to the centralized primary data store.

For each database configuration tested, every transactional workload was run three times in order to minimize the impact of any transient events in the cloud infrastructure. For each of these three "runs", the workload execution was repeated for every number of client threads (1, 2, 5, 10, 25, 50, 100, 200, 500, and 1000). This produced 30 separate output files (3 runs x 10 thread counts) for each workload. A data reduction program was developed to combine the measurements by averaging across the three runs for each thread count, and aggregating the results for all thread counts into a single file. The output of the data reduction program was a comma-separated variable (CSV) file that was structured to be imported into a Microsoft Excel spreadsheet template to produce formatted tables and graphical plots such as those shown below in "Results".

Our network partition tolerance testing was less automated. The YCSB test client was started, and then the Linux tc and iptables commands were manually invoked on the target server nodes to simulate network delays and partitions.

Results

We report results of evaluation of Transaction Performance, Partition Tolerance, and Data Model Mapping for three database products: MongoDB version 2.2, a document store (http://docs.mongodb.org/v2.2/); Cassandra version 2.0, a wide column or column family store (http://www.datastax.com/documentation/cassandra/2.0); and Riak version 1.4, a key-value store (http://docs.basho.com/riak/1.4.10/).

Transaction Performance Evaluation

Prototyping and evaluation were performed on two database server configurations: Single node server, and a nine-node configuration that was representative of a possible production deployment. The nine-node configuration used a topology that represented a geographically distributed deployment across three data centers. The data set was partitioned (i.e. "sharded") across three nodes, and then replicated to two additional groups of three nodes each. This was achieved using MongoDB's primary/secondary feature, and Cassandra's data center aware distribution feature. The Riak features did not allow this "3x3" data distribution approach, and so we used a configuration where the data set was sharded across all nine nodes, with three replicas of each shard stored on the same nine nodes. For each configuration, we report results for the three workloads discussed above.

The single node server configuration is not viable for production use: There is only one copy of each record, causing access bottlenecks that limit performance and a single point of failure limiting availability. However, this configuration provides some insights into the basic capabilities of each of the products tested. The throughput, in operations per second, is shown for each of the three workloads in Fig. 2 - Fig. 4. For the read-only workload, MongoDB achieved very high performance compared to Cassandra and Riak, due to MongoDB's indexing features that allowed the most recent observation record to be accessed directly, while Cassandra returned all observations for the selected patient back to the client where the most recent record was selected from the result set. Riak's relatively poor performance is due to an internal architecture that is not intended for deployment on a single node, as multiple instances of the storage backend (in our case, LevelDB) compete for disk I/O. For the write-only workload, Cassandra achieved a performance level comparable to the read-only throughput, while both MongoDB and Riak had showed lower write-only performance compared to read-only. Cassandra maintained a consistent performance level for the read/write workload, while both MongoDB and Riak showed a lower performance level. For the read/write workload, the average latency and 95th percentile latency is shown in Fig. 5 for read operations and in Fig. 6 for write operations. In both cases the average latency for both MongoDB and Riak increases as the number of concurrent client sessions increases.

Fig. 2. Throughput, Single Node, Read-only Workload

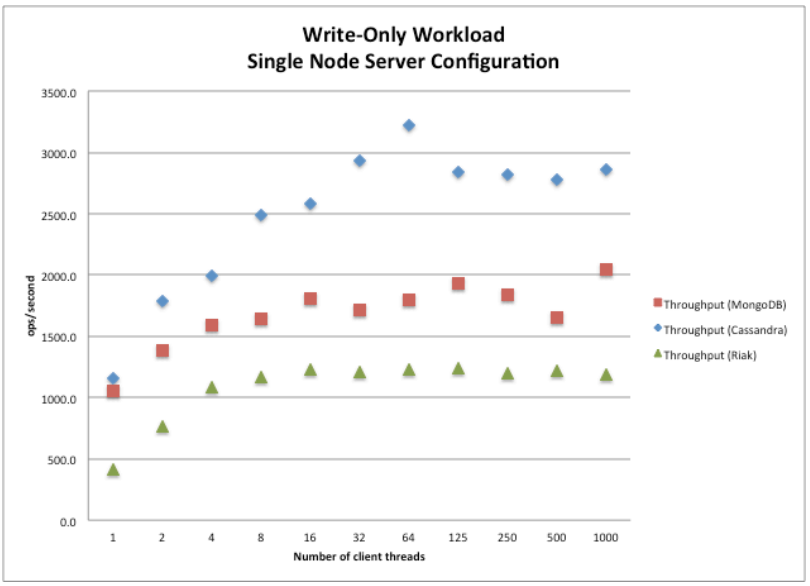

Fig. 3. Throughput, Single Node, Write-only Workload

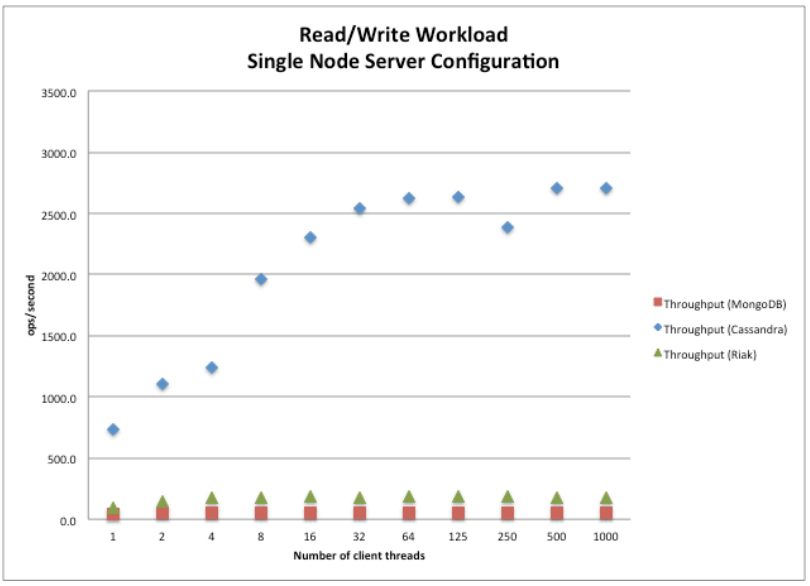

Fig. 4. Throughput, Single Node, Read/Write Workload

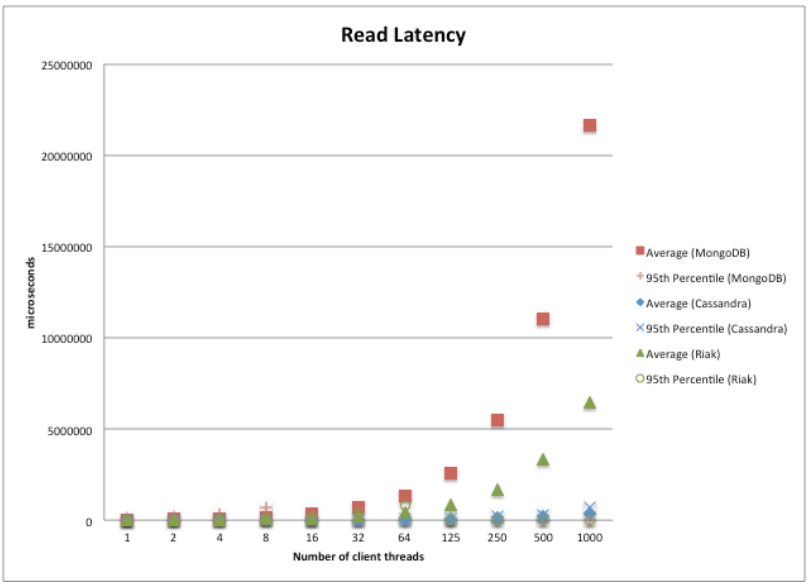

Fig. 5. Read Latency, Single Node, Read/Write Workload

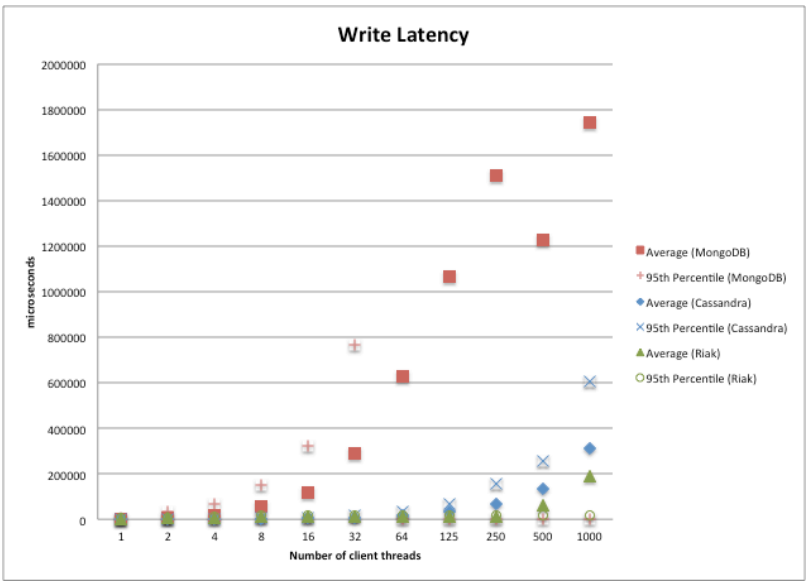

Fig. 6. Write Latency, Single Node, Read/Write Workload

Our testing varied the number of test client threads (or equivalently, the number of concurrent client sessions) from one up to 1000. Increasing the number of concurrent client sessions stresses the database server in several ways. There is an increase in network I/O and associated resource utilization (i.e. sockets and worker threads). There is also an increase in request rate, which increases utilization of memory, disk I/O, and other resources. The results in Fig. 2 - Fig. 4 show that server throughput increases with increased load, up to a point where I/O or resource utilization becomes saturated, and then performance remains flat as the load is increased further (for example, Cassandra performance at 64 concurrent sessions in Fig. 4), or decreases slightly, probably due to competition for resources within the server (for example, MongoDB performance at eight concurrent sessions in Fig. 4).

Operating with a large number of concurrent database client sessions is not typical for a NoSQL database. A typical NoSQL reference architecture has clients connecting first to a web server tier and/or an application server tier, which aggregates the client operations on the database using a pool of perhaps 16-64 concurrent sessions. However, our prototyping was in support of the modernization of a system that used thick clients with direct database connections, and so our customer wanted to understand the implications of retaining that thick client architecture.

In the nine-node configuration, we had to make several design decisions to define our representative production configuration. The first decision was how to distribute client connections across the server nodes. MongoDB uses a centralized router node, and all clients connected to the single router node. Cassandra's data center aware distribution feature created three sub-clusters of three nodes each, and client connections were spread uniformly across the three nodes in one of the sub-clusters. In the case of Riak, client connections were spread uniformly across the full set of nine nodes.

Another design decision was how to achieve strong consistency, which requires defining both write operation settings and read operation settings [Gorton 2014]. Each of the three databases offered slightly different options. The selected options are summarized in Table 1, with the details of the effect of each of the settings described in full in the product documentation for each of the databases.

Table 1. Write and read settings for representative production configuration

Database	Write Options	Read Options
MongoDB	Primary Acknowledged	Primary Preferred
Cassandra	EACH_ QUORUM	LOCAL_ QUORUM
Riak	quorum	Quorum

The throughput performance for the representative production configuration for each of the workloads is shown in Fig. 7 - Fig. 9. In all cases, Cassandra provided the best overall performance, with read-only workload performance roughly comparable to the single node configuration, and write-only and read/write workload performance slightly better than the single node configuration. This implies that, for Cassandra, the performance gains that accrue from decreased contention for disk I/O and other per node resources (compared to the single node configuration) are greater than the additional work of coordinating write and read quorums across replicas and data centers.

Riak performance in this representative production configuration is better than the single node configuration. In test runs using the write-only workload and the read/write workload, our test client had insufficient socket resources to execute the workload for 500 and 1000 concurrent sessions. These data points are reported as zero values in Fig. 8 and Fig. 9. We later determined that this resource exhaustion was due to an ambiguous description of Riak's internal thread pool configuration parameter, which creates a pool for *each* client session and not a pool shared by *all* client sessions. After determining that this did not impact the results for one through 250 concurrent sessions, and given that Riak had qualitative capability gaps with respect to our strong consistency requirements (as discussed below), we decided not to re-execute the tests for those data points.

MongoDB performance is significantly worse here than the single node configuration. Two factors influenced the MongoDB results. First, the representative production configuration is sharded, which introduces the router and configuration nodes into the MongoDB deployment architecture. The router node directs each request to the appropriate shard, based on key mapping information contained in the configuration node. Our tests ran with a single router node, which became a performance bottleneck. Fig. 10 and Fig. 11 show read and write operation latency for the read/write workload, with nearly constant average latency for MongoDB as the number of concurrent sessions is increased, which we attribute to saturation of the rapid saturation of the router node.

The second factor is the interaction between the sharding scheme used by MongoDB and the write-only and read/write workloads that we used, which was discovered near the end of our testing. Both Cassandra and Riak use a hash-based sharding scheme, which provides a uniformly distributed mapping from the range of keys onto the physical nodes. In contrast, MongoDB uses a range-based sharding scheme with rebalancing (http://docs.mongodb.org/v2.2/core/sharded-clusters/). Our write-only and read/write workloads generated a monotonically increasing sequential key for new records to be written, which caused all write operations to be directed to the same shard, since all of the write keys mapped into the space stored in that shard. This key generation approach is typical (in fact, many SQL databases have "autoincrement" key types that do this automatically), but in this case, it concentrates the write load for all new records in a single node and thus negatively impacting performance.

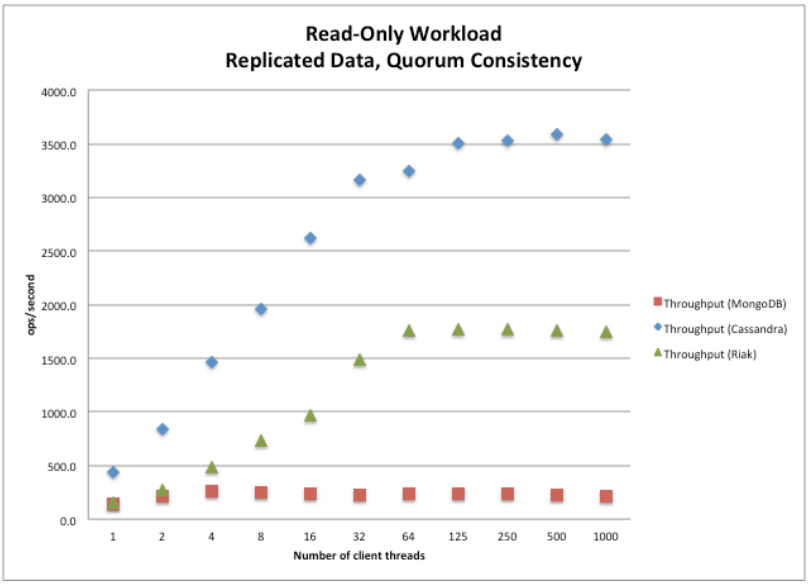

Fig. 7. Throughput, Representative Production Configuration, Read-Only Workload

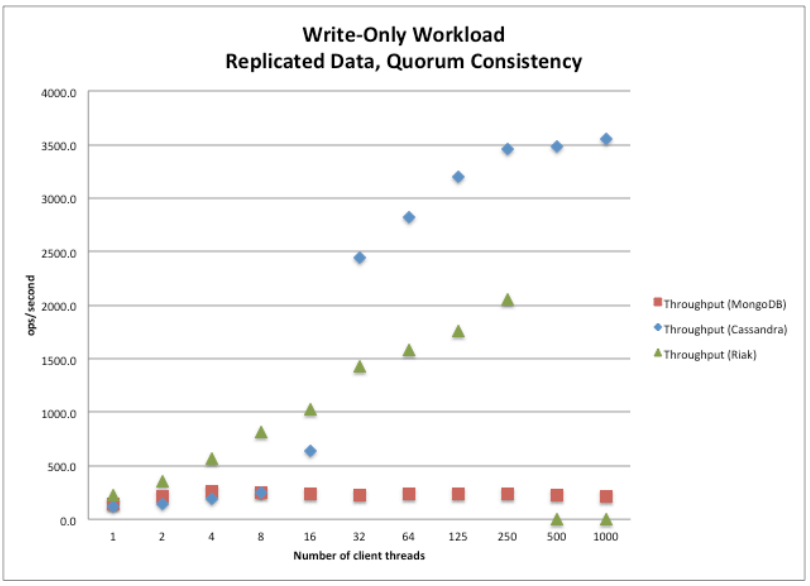

Fig. 8. Throughput, Representative Production Configuration, Write-Only Workload

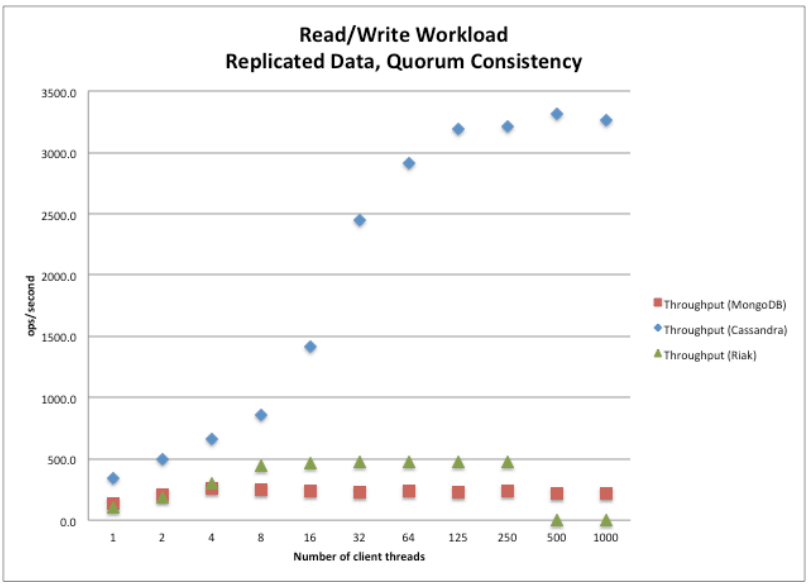

Fig. 9. Throughput, Representative Production Configuration, Read/Write Workload

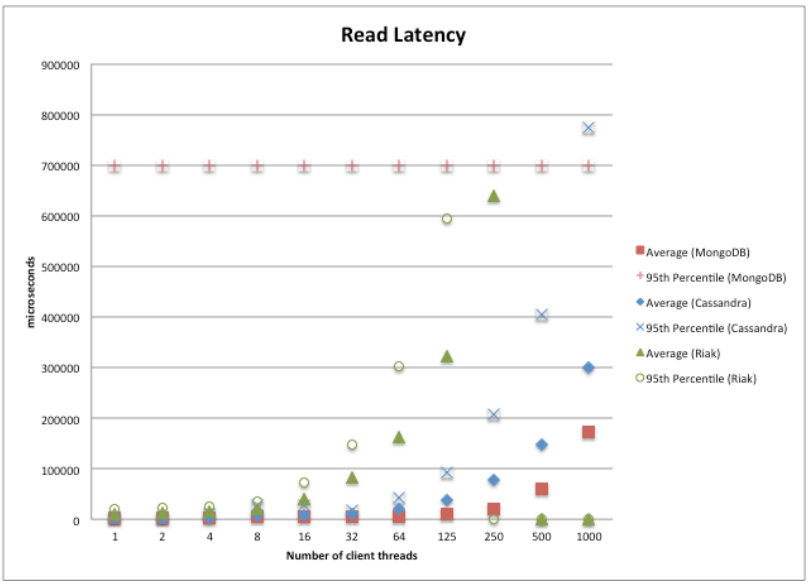

Fig. 10. Read Latency, Representative Production Configuration, Read/Write Workload

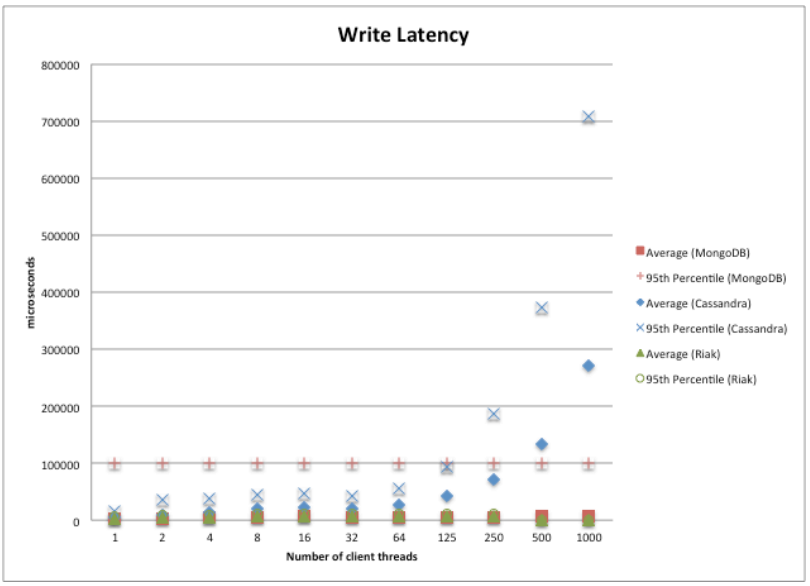

Fig. 11. Write Latency, Representative Production Configuration, Read/Write Workload

Finally we report transaction performance results that quantify the performance cost of strong replica consistency. These tests were limited to the Cassandra and Riak databases – the performance of MongoDB in the representative production configuration was such that no additional characterization of that database was warranted for our application. These tests used a combination of write operation settings and read operation settings that resulted in eventual consistency, rather than the strong consistency settings used in the tests described above. Again, each of the three databases offered slightly different options. The selected options are summarized in Table 2, with the details of the effect of each of the settings described in full in the product documentation for each of the databases.

Table 2. Write and read settings for eventual consistency configuration

Database	Write Options	Read Options
Cassandra	ONE	ONE
Riak	noquorum	noquorum

Fig. 12 shows throughput performance for the read/write workload on the Cassandra database, comparing the representative production configuration with the eventual consistency configuration. For any particular number of concurrent client sessions, the eventual consistency configuration provides higher throughput, and the eventual consistency configuration throughput flattens out at a higher level than strong consistency.

The same comparison is shown for the Riak database, in Fig. 13. Here, the difference in throughput between the strong consistency configuration and the eventual consistency configuration is much less obvious. As discussed above, test client configuration issues resulted in no data recorded for 500 and 1000 concurrent sessions.

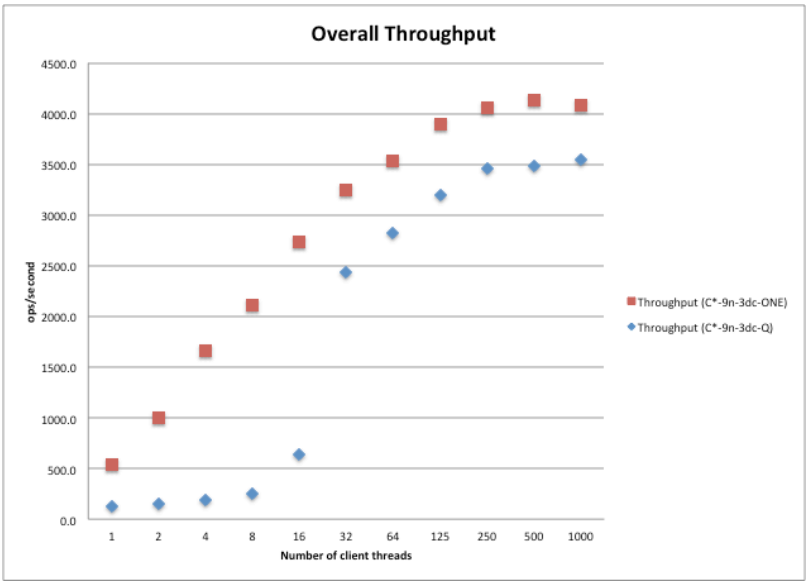

Fig. 12. Cassandra – Comparison of strong consistency and eventual consistency

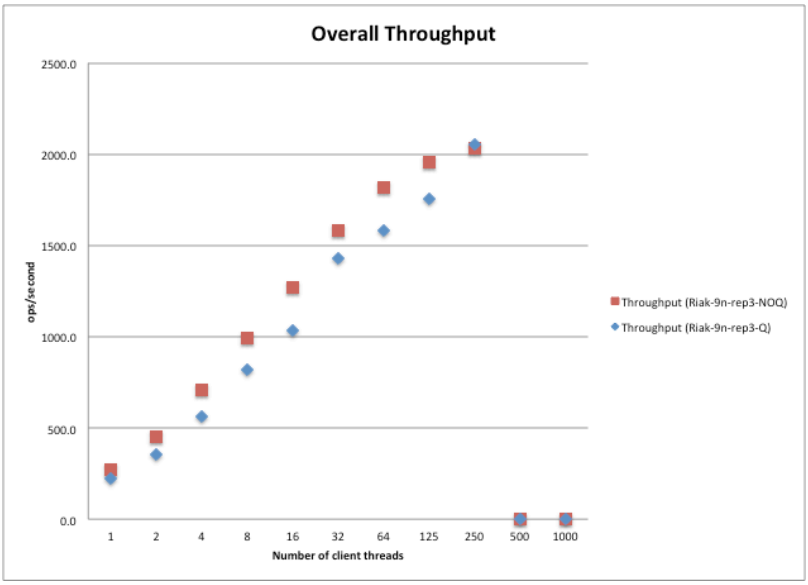

Fig. 13. Riak – Comparison of strong consistency and eventual consistency

In summary, the Cassandra database provided the best transaction throughput performance for our specific workloads and test configurations. We attribute this to several factors. First, hash-based sharding spread the request and storage load better than MongoDB. Second, the indexing features allowed efficient retrieval of the most recently written records. Finally, Cassandra's peer-to-peer architecture provides efficient coordination of both read and write operations across replicas and data centers.

Partition-tolerance

An important component of our stakeholder's environment is the presence of nodes (hospitals, clinics, etc.) with limited or poor network connectivity. The clinical sites are operating on networks with as little as T1 bandwidth (1.54 Mbps, while most consumer plans offer 10-15 Mbps), shared for all IT needs, including image transfers of X-Rays and the like.

The issues are both network throughput and latency—users are very sensitive to delays in reading and writing data using clinical applications like EHRs. We want to understand how NoSQL technologies will work in the presence of network problems, i.e. latency problems (L), node failures (NF), low bandwidth (BW), and packet loss (PL). There are two key questions. One, is there an effect on the database operations beyond that which is directly attributable to latency changes? Latency here refers solely to network latency, and is a subset of operation latency measured in YCSB. Two, are there any data integrity problems in a node-loss scenario? This is an important question, since a common tradeoff in NoSQL is strict consistency for availability. That is, NoSQL databases favor eventual consistency, so that the failure of a single node will not prevent writes from happening. Instead some form of global log is replicated, allowing failed nodes to re-sync with the history of the system. If a node fails, how much data is lost, and how long does it take the system to return to normal operations?

We constructed a sample network using some of these instances, and set up network simulations to reconstruct such problems. There are a number of tools that can simulate network problems: for example, netem and tc are tools that can force your internet connection into either thin, slow or lossy configurations.

We can perturb at least these variables:

- The client-data center connection or within datacenter. That is, tweak the client-database connection or tweak the network bandwidth between database instances (in a replica).
- The numbers of data center nodes lost (and which). Losing a primary instance might be worse than a secondary, in a replica.
- Length of node loss. How long is the node down? Does it come back, and then need to resynchronize the node list?
- Packet loss. How many (as a %)?
- Network delay (latency) or bandwidth (simulate with latency). We can add a throttle to make each packet take longer to leave the node (or arrive).

Here's an example of applying traffic control (tc) and using `ping` on a VPC machine:

```
<ping>
8 packets transmitted, 8 received, 0% packet loss, time 7212ms
rtt min/avg/max/mdev = 0.517/3.050/20.624/6.642 ms.
tc qdisc change dev eth0 root netem delay 100ms 10ms 25%
<ping>
25 packets transmitted, 25 received, 0% packet loss, time 24682ms
rtt min/avg/max/mdev = 91.254/103.790/199.620/20.311 ms
```

The bold numbers show the difference after applying a 100msec delay, with 10msec variance 25% of the time: greater than 3 fold increase in ping time (RTT).

The other tool we used was iptables, which is a method for manipulating firewall rules. For example, the command `iptables -I INPUT 1 -s 10.128.2.243 -j DROP` will add a rule at position 1 on INPUT to drop all packets from the .243 IP. This simulates a node failure on this node. In a replicated database, that might mean messages synchronizing writes between primary and secondary don't go through. In MongoDB, that triggers a rebalancing.

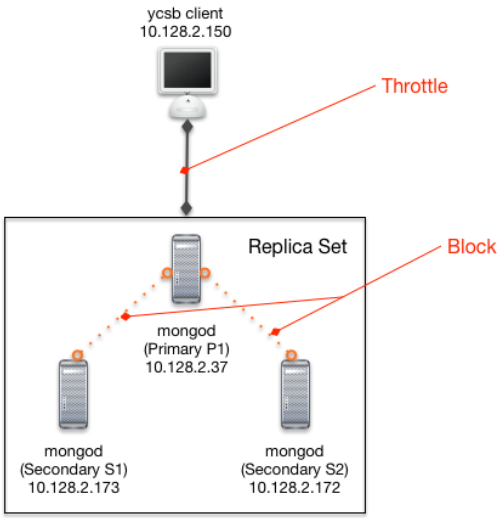

Fig. 14. Network configuration for partition testing of MongoDB

Our network configuration for Mongo is shown in Fig. 14. We can also configure MongoDB with different settings for read preference (what node can be read from) and write concern (as described in the MongoDB product documentation), or how important it is that our recent write operation is stored on all nodes in the replica. In these experiments we used read preference of primary and write concerns of both acknowledged (Fig. 15) and *replica acknowledged*[1], where the data must first be written to the replica prior to the write being acknowledged to the client.

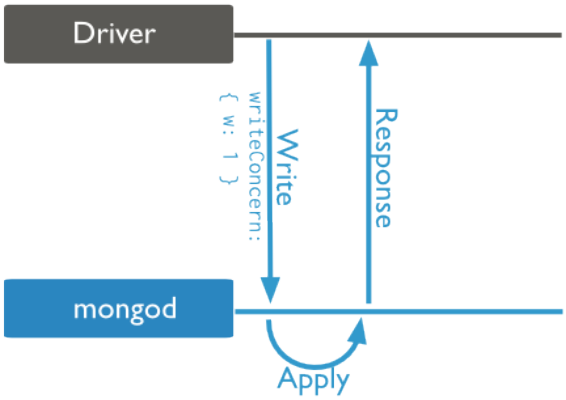

Fig. 15. MongoDB "acknowledged" write concern

We conducted the experiments only on MongoDB, and report here on the read-write scenarios with 80% reads. Our baseline was a series of workloads run with no network perturbations deliberately introduced, and although Amazon's network topology is naturally subject to the stochastic changes inherent in the wider Internet, we did not observe any major disturbances. Our second run (*no-ack*) introduced some node loss at random intervals, and our third run (*ack)* introduced similar stochastic node loss using a write concern of 'secondary acknowledged'. We observed obvious short-term spikes in the last two runs, but this is likely due to the Java garbage collector being activated. There was no clear causal relationship between a node failing, and subsequent throughput problems. However, as can be seen in Fig. 16, there were clear overhead burdens in moving to this level of consistency guarantee. Furthermore, there are large numbers of exceptions that must be handled because the connection in the Mongo client does not handle them by default, and cannot detect a problem until invoked.[2] For example, in the *no-ack* scenario, we do not have any faults detected doing a write operation, while in the *ack* scenario, between 17% and 34% of writes would fail, due to the node loss.

[1] http://docs.mongodb.org/manual/core/replica-set-write-concern/

[2] http://stackoverflow.com/questions/18564607/com-mongodb-dbportpool-goterror-warning-emptying-dbportpool-to-ip27017-b-c-of

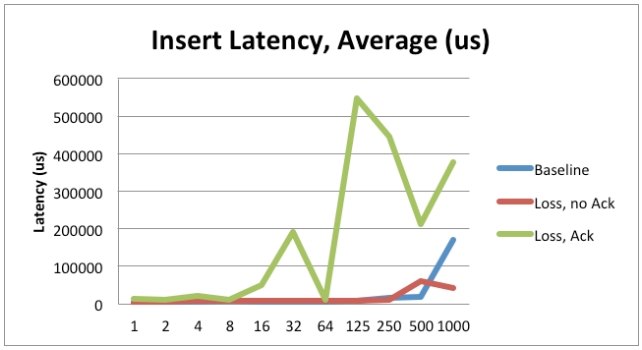

Fig. 16. Impact of partition on MongoDB insert operation latency

We also examined the results of throttling the bandwidth with respect to a high bandwidth situation. Using the tc tool, we applied throttling to simulate latency on the network of +100 ms 25% autocorrelated with +/- 10% variance. As expected, this also slows down the throughput of our MongoDB client, as shown in Fig. 17 (higher is better). By 250 threads, however, the concurrency of the threads has overwhelmed our network link in both cases, resulting in a very similar performance.

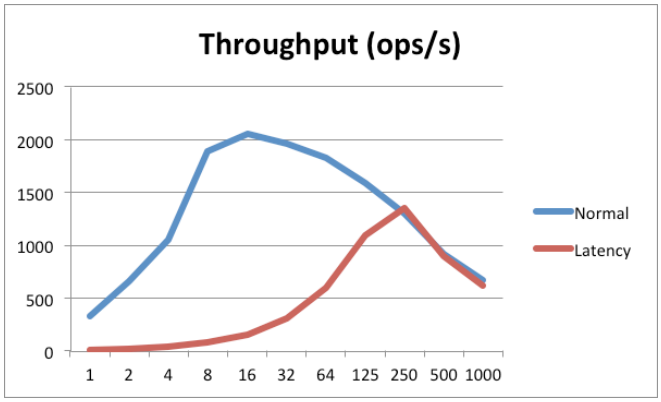

Fig. 17. Impact of partition on MongoDB insert operation throughput

Finally, we investigated the possibility of data loss. If writes are not acknowledged by the replicas, there is a chance of records being overwritten when a node that went down is brought back online. We recorded the number of records written on all three nodes, as well as the timestamps of the *oplog*, the central data structure for ensuring consistency in Mongo. We could not see any major differences in these values, and in particular, the number of nodes was the same in all three scenarios. For example, a result like:

```
37: optime=Timestamp(1411011459, 11) records: 5084075
173: optime=Timestamp(1411011459, 18) records: 5084075
```

tells us that the primary, node 37, has the same number of records as the secondary.

Our conclusions from the latency/consistency experiments in MongoDB was that the database performed as expected, handling node loss gracefully enough. However, the large number of exceptions generated in a high-consistency (secondary acknowledge) scenario puts a large burden on the programmer to determine correct behavior (likely to re-send the data).

Data Model Mapping

Throughout the prototype design and development, we developed a set of findings that are qualitative. Here we report on these qualitative findings in the area of alignment of our data model with the capabilities provided by each database.

The most significant data modeling challenge was the representation of the one-to-many relation from patient to lab results, coupled with the need to efficiently access the most-recently written lab results for a particular patient.

Zola has analyzed the various approaches and tradeoffs of representing the one-to-many relation in MongoDB [Zola 2014]. We used a composite index of (Patient ID, Observation ID) for lab result records, and also indexed by the lab result date-time stamp. This allowed efficient retrieval of the most recent lab result records for a particular patient.

A similar approach was used for Cassandra. Here we used a composite index of (PatientID, lab result date-time stamp). This caused the result set returned by the query to be ordered, making it efficient to find and filter the most recent lab records for a particular patient. Cassandra 2.0, which was released after our testing was completed, offers the ability to iterate over the result set, returning only one record at a time to the client, which may provide additional efficiency improvements.

Representing the one-to-many relation in Riak was more complicated. Riak's basic capability is retrieval of a value, given a unique key. The peer-to-peer gossip protocol ensures that every node in the cluster can map from a record's key to the node (or nodes, if data is replicated) that holds the record. Riak also provides a "secondary index" capability that allows record retrieval when the key is not known, however each server node in the cluster stores only the indexes for the portion of the records that are managed by the node. When an operation requests all records with a particular secondary index value, the request coordinator must perform a "scatter-gather", asking storage node for records with the desired secondary index value, waiting for all nodes to respond, and then sending the list of keys back to the requester. The requester must then make a second database request with the list of keys, in order to retrieve the record values. The latency of the "scatter-gather" to locate records, and the need for two request/response round trips had a negative impact on Riak's performance for our data model. Furthermore, there is no mechanism for the server to filter and return only the most recent observations for a patient. All observations must be returned to the client, and then sorted and filtered. We attempted to de-normalize further, by introducing a data set where each record contained a list of the most recently written observations for each patient. However, this required an atomic read-modify-write of that list every time a new observation is added for the patient, and that capability was not supported in Riak 1.4.

We also performed a systematic categorization of the major features available in the 3 databases for issuing queries. For each feature, we classified the capabilities of each database against a set of criteria so that they can be directly compared. Table 3 shows the results of this comparison.

Table 3. Comparing the Data Querying Capabilities for Cassandra, MongoDB, and Riak

	API-Based	declarative queries	REST/HTTP-based	Cursor-based queries	JOIN-style queries	Restrict number of returned objects	Expire data values	Languages supported	Complex data types	Key matching options	Sorting of query results	Triggers
Cassandra Query Language Features	supported	supported	not suppported	supported	not suppported	supported	supported	Java C# C/C++ Erlang	sets nested structures arrays	exact partial match	ascending descending	pre-commit
MongoDB Query Language Features	supported	not suppported	not suppported	supported	not suppported	supported	supported	Java C# Python C/C++ Perl PHP Ruby Scala	maps nested structures arrays	exact partial match wildcards regular expressions	ascending descending	not supported
Riak Query Language Features	supported	not suppported	supported	supported	not suppported	supported	supported	Java C# PHP Ruby Erlang	lists maps sets arrays	exact	ascending	pre-commit post-commit

This comparison allows our customer to evaluate the databases against their specific needs, both for runtime and software development (e.g. programming language support, data type support). Both these sets of requirements ultimately are important in any technology adoption decision, and must be weighted appropriately to best satisfy organizational requirements.

Objectively, MongoDB and Cassandra both provided a relatively straight-forward data model mapping and both provided the strong consistency needed for our customer's EHR application. Subjectively, the data model mapping in MongoDB was more transparent than the use of the Cassandra Query Language (CQL), and the indexing capabilities of MongoDB were a better fit for this application.

Lessons learned

We separate our lessons learned into two broad categories. The first category pertains to the overall technology selection process for the storage layer in a big data system, and the second category pertains to the development and execution of the prototyping and measurement.

Technology Selection Process

In big data systems using NoSQL technology, there is a coupling of concerns between the selected storage layer product, the deployment topology, and the application architecture [Gorton 2014]. Technology selection is an architecture decision that must be made early in the design cycle, and is difficult and expensive to change. The selection must be made in a setting where the problem space definition may be incomplete, and the solution space is large and rapidly changing as the open source landscape continues to evolve.

We found it helpful to employ a process that considers the candidate technology in the context of use. In the case of NoSQL and big data systems, the context of use includes characterizing the size and growth rate of the data (number of records and record size), the complexity of the data model along with concrete of key relations and navigations, operational environment including system management practices and tools, and user access patterns including operation mix, queries, and number of concurrent users.

We found the use of quality attribute scenarios [Barbacci 2003] to be helpful to assist stakeholders to concretely define selection criteria. After eliciting a number of scenarios of interest to one or more stakeholders, we were able to cluster and prioritize the scenarios to identify "go/no-go" criteria – the capabilities that the selected technology must have, or behaviors or features that, if present in a product, disqualify it from use in the system. These criteria may be quantitative or qualitative. In the case of quantitative criteria, there may be hard thresholds that must be met, but this type of criterion can be problematic to validate, since there are a large number of infrastructure, operating system, database product, and test client parameters that can be tuned in combination to impact absolute performance (and certainly, a final architecture design must include that tuning). It can be more useful for product selection to frame the criterion in terms of concerns about the shape of the performance curve, for example, does throughput increase linearly with load up to some level where throughput flattens out, and is that point of flattening within our range of interest? Understanding the sensitivities and trade offs in a product's capabilities may be sufficient to make a selection, and also provides valuable information to make downstream architecture design decisions regarding the selected product.

We used this context of use definition perform a manual survey of product documentation to identify viable candidate products. As this was our first time working with NoSQL technology, the manual survey process was slow and inefficient. We began to collect and aggregate product feature and capability information into a queryable, reusable knowledge base, which included general quality attribute scenarios as templates for concrete scenarios, and linked the quality attribute scenarios to particular product implementations and features. This knowledge base was used successfully for later projects, and is an area for further research.

The selection process must balance cost (in time and resources) with fidelity and measurement precision. The solution space is changing rapidly. During the course of our evaluation, each of the candidate products released at least one new version that included changes to relevant features, so a lengthy evaluation process is likely to produce results that are not relevant or valid. Furthermore, if a cloud infrastructure is used to support the prototyping and measurement, then changes to that environment can impact results. For example, during our testing process, Amazon changed standard instance types offered in EC2. Our prototypes all used the instance type "m1.large", which was eliminated as a standard offering during our testing, but we were still able to use it as a "legacy instance type" until completion of our tests. Our recommendation is to perform prototyping and measurement for just two or three products, in order to complete quickly and deliver valid and relevant results.

To maintain urgency and forward progress while defining the selection criteria and performing the prototyping and measurement, we found it useful to ask, "How will knowing the answer to this question change the final selection decision?" In many cases, the answer would be needed to make downstream design decisions of the particular product was to be used in the system, but the answer was not necessary to make a selection decision, and so that measurement was put off until an appropriate time later in the design process.

Prototyping and Measurement

We started our prototyping and measurement using a cloud infrastructure, which proved to be essential for efficient management and execution of the tests. Although our initial test was a single server and single client, we quickly grew to product configurations with more than 10 servers, and were frequently executing on more than one product configuration at a time. Our peak utilization was over 50 concurrently executing server nodes (divided across several product configurations), which is more than can be efficiently managed in most physical hardware environments.

We had a continual tension between using manual processes for server deployment and management, and automating some or all of these processes. Repeating manual tasks conflicts with software engineering best practices such as "don't repeat yourself" (http://c2.com/cgi/wiki?DontRepeatYourself), but in retrospect we think that the decision to always make slow forward progress, rather than stopping to automate, was appropriate. Organizations that already have a proven automation capability and expertise in place may reach a different conclusion. We did develop simple scripts to automate test execution and most of the data collection, processing, and visualization. These tasks were performed frequently, had many steps, and needed to be repeatable.

We started evaluating each product using a single server configuration, which validated the software installation and configuration, and allowed easier debugging of the test client. Recognize that some NoSQL products (e.g., Riak) are not designed to operate well in this configuration, and so this configuration may not produce any useful test results. We next expanded our configuration to partition ("shard") the data set across multiple nodes, which allowed us to validate the cluster configuration. Finally, we introduced data replication into the test configuration. We found this sequence to be effective, providing confidence in the validity of the measurements.

Our context of use and workloads required that the database was populated with a complete data set. We found that bulk load is a special type of workload, and each database product had specific recommendations and special APIs for this function. In some cases (i.e. MongoDB), recommendations like "pre-splitting" the data set simply improved bulk load performance. In other cases, we found that following the recommendations was necessary to avoid failures due to resource exhaustion in the database server. We recommend that if bulk load is not one of your selection criteria, then take a brute force approach to load the data, and then use database backups, or virtual machine or storage volume snapshots to return to the initial state as needed.

All of our tests that performed write operations ended the test by restoring the database to its initial state. We found that deleting records in most NoSQL databases is very slow, taking as much as 10 times longer than a read or write operation. In retrospect, we would consider using snapshots to restore state, rather than cleaning up using delete operations.

It is critical that you understand your measurement framework. Although YCSB has become the *de facto* standard for NoSQL database characterization, the 95th and 99th percentile measurements that it reports are only valid under certain latency conditions, as we discussed above. The YCSB implementation could be modified to extend the validity of those measurements to a broader range of latencies, or alternative metrics can be used for selection criteria.

Related Work

Benchmarking of databases is generally based on the execution of a specific workload against a specific data set, such as the Wisconsin benchmark for general SQL processing [Bitton] or the TPC-B benchmark for transaction processing [Anon], with system and infrastructure configuration left as decisions for the evaluator. These publically available workload definitions have long enabled vendors and others to publish measurements, which consumers can then attempt to map to the target workload, configuration, and infrastructure for product comparison and selection. These benchmarks were developed for relational data models, and are not relevant for NoSQL systems. The default data sets and workloads YCSB [Cooper] has emerged as a commonly used benchmark for NoSQL systems. (Note that although we used the YCSB framework, we substituted a data set and workload definition that was specific to our system requirements.) YCSB++ [Patil] extends YCSB with more sophisticated, multi-phase workload definitions and support for multiple coordinated clients to increase the load on the database server. There is an emerging collection of published measurements using YCSB and YCSB++, from product vendors [Datastax, Nelubin] and from researchers [Abramova, Floratau].

All of the work discussed above has used "generic" data sets and workloads. In contrast, the T-check method [Lewis] performs evaluation by defining selection criteria and then prototyping and measuring the candidate technology in the context of use, and was an influence on our creation of the LEAP4BD method. The work reported by Dede and colleagues [Dede 2013] is an example of the type of hybrid qualitative and quantitative analysis that we performed, using system-specific data sets and workloads to answer specific questions about the candidate technologies within a particular context of use.

Conclusion

NoSQL database technology offers benefits of scalability and availability through horizontal scaling, replication, and simplified data models, but the specific implementation must be chosen early in the architecture design process. We have described a systematic method to perform this technology selection in a context where the solution space is broad and changing fast, and the system requirements may not be fully defined. Our method evaluates the products in the specific

context of use, starting with elicitation of quality attribute scenarios to capture key architecture drivers and selection criteria. Next, product documentation is surveyed to identify viable candidate technologies, and finally, rigorous prototyping and measurement is performed on a small number of candidates to collect data to make the final selection.

We described the execution of this method to evaluate NoSQL technologies for an electronic healthcare record system, and present the results of our measurements of transactional performance and partition tolerance, along with a qualitative assessment of alignment of the NoSQL data model with system-specific requirements. We present lessons learned from our application of the selection method, and from our execution of the prototyping and measurements.

We have identified the benefits of having a trusted knowledge base that can be queried to discover the features and capabilities of particular NoSQL products, and accelerate the initial screening to identify viable candidate products for a particular set of quality attribute scenario requirements. This is an area for further research. Additionally, our testing of network partitions required manual intervention, and further research to automate the triggering of partitions and measurement collection is needed.

Acknowledgements

Copyright 2014 ACM.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

This material has been approved for public release and unlimited distribution.

T-Check[SM]

DM-0001661

References

[1] D. Agrawal, S. Das, and A. El Abbadi, “Big data and cloud computing: current state and future opportunities,” in *Proc. 14th International Conference on Extending Database Technology*, 2011, pp. 530--533.

[2] F. Chang, J. Dean, S. Ghemawat, et al., “Bigtable: A Distributed Storage System for Structured Data,” *ACM Transactions on Computing Systems,* vol. 26, no. 2, 2008.

[3] G. DeCandia, D. Hastorun, M. Jampani, et al., “Dynamo: Amazon's Highly Available Key-value Store,” in *Proc. Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, USA, 2007, pp. 205-220. doi: 10.1145/1294261.1294281

[4] P. J. Sadalage and M. Fowler, *NoSQL Distilled.* Addison-Wesley Professional, 2012.

[5] E. Brewer, “CAP twelve years later: How the "rules" have changed,” *Computer,* vol. 45, no. 2, pp. 23-29, 2012.

[6] D. J. Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story,” *Computer,* vol. 45, no. 2, pp. 37-42, 2012. doi: 10.1109/MC.2012.33

[7] M. R. Barbacci, R. J. Ellison, A. J. Lattanze, et al., “Quality Attribute Workshops (QAWs).” Software Engineering Institute, Technical Report, CMU/SEI-2003-TR-016, 2003, http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=6687.

[8] D. Bitton, D. J. DeWitt, and C. Turbyfill, “Benchmarking Database Systems A Systematic Approach,” in *Proc. 9th International Conference on Very Large Data Bases (VLDB '83)*, 1983, pp. 8-19.

[9] B. F. Cooper, A. Silberstein, E. Tam, et al., “Benchmarking Cloud Serving Systems with YCSB,” in *Proc. 1st ACM Symposium on Cloud Computing (SoCC '10)*, 2010, pp. 143-154. doi: 10.1145/1807128.1807152

[10] Datastax, “Benchmarking Top NoSQL Databases.” Datastax Corporation, White Paper, 2013.

[11] I. Gorton, A. Liu, and P. Brebner, “Rigorous evaluation of COTS middleware technology,” *Computer,* vol. 36, no. 3, pp. 50-55, 2003.

[12] I. Gorton and J. Klein, “Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems,” *IEEE Software,* vol. PP, no. 99, 18 March 2014. doi: 10.1109/MS.2014.51

[13] M. Finnegan, “Boeing 787s to create half a terabyte of data per flight, says Virgin Atlantic,” *Computerworld UK,* 6 March 2013, http://www.computerworlduk.com/news/infrastructure/3433595/boeing-787s-to-create-half-a-terabyte-of-data-per-flight-says-virgin-atlantic/ (Accessed 20 Feb 2014).

[14] P. Groves, B. Kayyali, D. Knott, et al., “The `big data' revolution in healthcare.” McKinsey & Company, Report, 2013, http://www.mckinsey.com/insights/health_systems_and_services/the_big-data_revolution_in_us_health_care (Accessed 20 Feb 2014).

[15] M. Armbrust "A View of Cloud Computing", *Comm. ACM*, vol. 53, no. 4, pp. 50-58, 2010.

[16] W. Zola. *6 Rules of Thumb for MongoDB Schema Design: Part 1* [Online]. http://blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1 (Accessed 18 Sep 2014).

[17] Health Level 7. *Fast Healthcare Interoperability Resources (FHIR) Specification*, Draft Standard for Trial Use (DSTU), version 0.80-2325, 3 April 2014. http://www.hl7.org/implement/standards/fhir/

[18] D. Bitton, D. J. DeWitt, and C. Turbyfill, “Benchmarking Database Systems A Systematic Approach,” in *Proc. 9th International Conference on Very Large Data Bases (VLDB '83)*, 1983, pp. 8-19.

[19] Anon, D. Bitton, M. Brown, et al., “A Measure of Transaction Processing Power,” *Datamation,* vol. 31, no. 7, pp. 112-118, April 1985.

[20] S. Patil, M. Polte, K. Ren, et al., “YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores,” in *Proc. 2nd ACM Symposium on Cloud Computing (SOCC '11)*, 2011, pp. 9:1--9:14. doi: 10.1145/2038916.2038925

[21] D. Nelubin and B. Engber, “Ultra-High Performance NoSQL Benchmarking: Analyzing Durability and Performance Tradeoffs.” Thumbtack Technology, Inc., White Paper, 2013.

[22] V. Abramova and J. Bernardino, “NoSQL Databases: MongoDB vs Cassandra,” in *Proc. International C* Conference on Computer Science and Software Engineering (C3S2E '13)*, 2013, pp. 14-22. doi: 10.1145/2494444.2494447

[23] A. Floratou, N. Teletia, D. J. DeWitt, et al., “Can the Elephants Handle the NoSQL Onslaught?,” *Proceedings of the VLDB Endowment,* vol. 5, no. 12, pp. 1712-1723, 2012.

[24] G. A. Lewis and L. Wrag, “A Process for Context-Based Technology Evaluation.” Carnegie Mellon Software Engineering Institute, Technical Note, CMU/SEI-2005-TN-025, 2005.

[25] E. Dede, M. Govindaraju, D. Gunter, et al., “Performance Evaluation of a MongoDB and Hadoop Platform for Scientific Data Analysis,” in *Proc. 4th ACM Workshop on Scientific Cloud Computing (Science Cloud '13)*, 2013, pp. 13-20. doi: 10.1145/2465848.2465849